Package: rupturesRcpp (via r-universe)

October 11, 2026

Type Package

Title Object-Oriented Interface for Offline Change-Point Detection

Version 2.0.0

Description A collection of efficient implementations of popular offline change-point detection algorithms, featuring a consistent, object-oriented interface for practical use.

Encoding UTF-8

URL https://edelweiss611428.github.io/rupturesRcpp/, https://github.com/edelweiss611428/rupturesRcpp

BugReports https://github.com/edelweiss611428/rupturesRcpp/issues

RoxygenNote 7.3.3

License MIT + file LICENSE

LinkingTo Rcpp, RcppArmadillo

Imports Rcpp, R6, ggplot2, patchwork, methods

Suggests testthat (>= 3.0.0), reticulate, binsegRcpp, covr

Config/testthat/edition 3

Collate 'costFuncR6.R' 'DynpR6.R' 'PeltR6.R' 'WindowR6.R' 'binSegR6.R' 'costFactoryR6.R' 'rupturesRcpp-package.R' 'zzz.R'

NeedsCompilation yes

Author Minh Long Nguyen [aut, cre], Toby Hocking [aut], Charles Truong [aut], Huy Nhat Minh Nguyen [ctb]

Maintainer Minh Long Nguyen <edelweiss611428@gmail.com>

Repository https://cran.r-universe.dev

Date/Publication 2026-10-11 15:00:02 UTC

RemoteUrl https://github.com/cran/rupturesRcpp

RemoteRef HEAD

RemoteSha 6ba924dc34e1c071565aedb2863ceb012107a31c

Contents

binSeg 2
costFactory 9
costFunc 12
Dynp 16
PELT 22
Window 28

Index **37**

`binSeg` *Binary Segmentation (*`binSeg`*)*

Description

An `R6` class implementing binary segmentation for offline change-point detection.

Details

Binary segmentation is a classic algorithm for change-point detection that recursively splits the data at locations that minimise the cost function.

`binSeg` requires a `R6` object of class `costFunc`, which can be created via `costFunc$new()`. Currently, the following cost functions are supported:

- `"L1"` and `"L2"` for (independent) piecewise Gaussian process with **constant variance**
- `"SIGMA"`: for (independent) piecewise Gaussian process with **varying variance**
- `"VAR"`: for piecewise Gaussian vector-regressive process with **constant noise variance**
- `"LinearL2"`: for piecewise linear regression process with **constant noise variance**
- `"LinearSIGMA"`: for piecewise linear regression process with **varying noise covariance**
- `"LinearL1"`: for piecewise linear regression process under **L1 (least absolute deviations) loss**

See `$eval()` method for more details on computation of cost.

Some examples are provided below. See the package website for detailed usage!

Methods

`$new()` Initialises a `binSeg` object.

`$describe()` Describes the `binSeg` object.

`$fit()` Constructs a `binSeg` module in `C++`.

`$eval()` Evaluates the cost of a segment.

`$predict()` Performs `binSeg` given a linear penalty value.

`$segments()` Returns the cost and parameter estimates of each segment from the latest `$predict()`.

`$plot()` Plots change-point segmentation in `ggplot` style.

`$clone()` Clones the `R6` object.

Active bindings

`minSize` Integer. Minimum allowed segment length. Can be accessed or modified via `$minSize`. Modifying `minSize` will automatically trigger `$fit()`.

`jump` Integer. Search grid step size. Can be accessed or modified via `$jump`. Modifying `jump` will automatically trigger `$fit()`.

`costFunc` `R6` object of class `costFunc`. Search grid step size. Can be accessed or modified via `$costFunc`. Modifying `costFunc` will automatically trigger `$fit()`.

`tsMat` Numeric matrix. Input time series matrix of size $n \times p$. Can be accessed or modified via `$tsMat`. Modifying `tsMat` will automatically trigger `$fit()`.

`covariates` Numeric matrix. Input time series matrix having a similar number of observations as `tsMat`. Can be accessed or modified via `$covariates`. Modifying `covariates` will automatically trigger `$fit()`.

Methods

Public methods:

- `binSeg$new()`
- `binSeg$describe()`
- `binSeg$fit()`
- `binSeg$eval()`
- `binSeg$predict()`
- `binSeg$segments()`
- `binSeg$getHistory()`
- `binSeg$plotElbow()`
- `binSeg$plot()`
- `binSeg$clone()`

Method `new()`**:** Initialises a `binSeg` object.

Usage:

```
binSeg$new(minSize, jump, costFunc)
```

Arguments:

`minSize` Integer. Minimum allowed segment length. Default: `1L`.

`jump` Integer. Search grid step size: only positions in {k, 2k, ...} are considered. Default: `1L`.

`costFunc` A `R6` object of class `costFunc`. Should be created via `costFunc$new()` to avoid error. Default: `costFunc$new("L2")`.

Returns: Invisibly returns `NULL`.

Method `describe()`**:** Describes a `binSeg` object.

Usage:

```
binSeg$describe(printConfig = FALSE)
```

Arguments:

`printConfig` Logical. Whether to print object configurations. Default: `FALSE`.

Returns: Invisibly returns a list storing at least the following fields:

`minSize` Minimum allowed segment length.

`jump` Search grid step size.

`costFunc` The `costFunc` object.

`fitted` Whether or not `$fit()` has been run.

`tsMat` Time series matrix.

`covariates` Covariate matrix (if exists).

`n` Number of observations.

`p` Number of features.

Method `fit()`**:** Constructs a `C++` module for binary segmentation.

Usage:

```
binSeg$fit(tsMat = NULL, covariates = NULL)
```

Arguments:

`tsMat` Numeric matrix. A time series matrix of size $n \times p$ whose rows are observations ordered in time. If `tsMat = NULL`, the method will use the previously assigned `tsMat` (e.g., set via the active binding `$tsMat` or from a prior `$fit(tsMat)`). Default: `NULL`.

`covariates` Numeric matrix. A time series matrix having a similar number of observations as `tsMat`. Required for models involving both dependent and independent variables. If `covariates = NULL` and no prior covariates were set (i.e., `$covariates` is still `NULL`), the model is force-fitted with only an intercept. Default: `NULL`.

Details: This method constructs a `C++ binSeg` module and sets `private$.fitted` to `TRUE`, enabling the use of `$predict()` and `$eval()`. Some precomputations are performed to allow `$predict()` to run in linear time with respect to the number of data points (see `$predict()` for more details).

Returns: Invisibly returns `NULL`.

Method `eval()`**:** Evaluate the cost of the segment (a,b]

Usage:

```
binSeg$eval(a, b)
```

Arguments:

`a` Integer. Start index of the segment (exclusive). Must satisfy `start < end`.

`b` Integer. End index of the segment (inclusive).

Details: The segment cost is evaluated as follows:

- **L1 cost function**:

$$c_{L_1}(y_{(a+1)\dots b}) := \sum_{t=a+1}^{b} \|y_t - \tilde{y}_{(a+1)\dots b}\|_1$$

  where $\tilde{y}_{(a+1)\dots b}$ is the coordinate-wise median of the segment. If $a \geq b - 1$, return 0.

- **L2 cost function**:

$$c_{L_2}(y_{(a+1)\dots b}) := \sum_{t=a+1}^{b} \|y_t - \bar{y}_{(a+1)\dots b}\|_2^2$$

  where $\bar{y}_{(a+1)\dots b}$ is the empirical mean of the segment. If $a \geq b - 1$, return 0.

- **SIGMA cost function**:

$$c_{\Sigma}(y_{(a+1)\dots b}) := (b-a)\log\det\hat{\Sigma}_{(a+1)\dots b}$$

where $\hat{\Sigma}_{(a+1)\dots b}$ is the empirical covariance matrix of the segment without Bessel's correction. Here, if `addSmallDiag = TRUE`, a small bias `epsilon` is added to the diagonal of estimated covariance matrices to improve numerical stability.

By default, `addSmallDiag = TRUE` and `epsilon = 1e-6`. In case `addSmallDiag = TRUE`, if the covariance matrix is numerically singular (its log-determinant cannot be computed) or its log-determinant is smaller than the lower bound `p*log(epsilon)`, return `(b - a)*p*log(epsilon)`, otherwise, output an error message.

- **VAR(r) cost function**:

$$c_{\mathrm{VAR}}(y_{(a+1)\dots b}) := \sum_{t=\max(a,r)+1}^{b} \left\| y_t - \hat{c} - \sum_{j=1}^{r} \hat{A}_j y_{t-j} \right\|_2^2$$

where $\hat{c}$ and $\hat{A}_j$ are the OLS estimates of the intercept and VAR coefficients on the segment. The lagged values y_{t-j} may come from before $a+1$, so only the first r observations of the whole series lack a full set of lags. If the system is singular, an approximate (minimum-norm) least-squares solve is used. If $b - a < p * r + 1$ (i.e., not enough observations), or $a > n - r$ (where `n` is the time series length), return 0.

- **"LinearL2"** for piecewise linear regression process with **constant noise variance**

$$c_{\text{LinearL2}}(y_{(a+1):b}) := \sum_{t=a+1}^{b} \|y_t - X_t\hat{\beta}\|_2^2$$

where $\hat{\beta}$ are OLS estimates on segment $(a+1):b$. If segment is shorter than the minimum number of points needed for OLS, return 0.

- **"LinearSIGMA"** for piecewise linear regression process with **varying noise covariance**

$$c_{\text{LinearSIGMA}}(y_{(a+1):b}) := (b-a)\log\det\hat{\Sigma}_{(a+1):b}$$

where $\hat{\Sigma}_{(a+1):b}$ is the empirical covariance matrix of OLS residuals $y - X\hat{\beta}$ on segment $(a+1):b$, divided by $b-a$ with no degrees-of-freedom correction for the fitted coefficients, and otherwise estimated the same way as in the SIGMA cost function.

- **"LinearL1"** for piecewise linear regression process under **L1 (least absolute deviations) loss**

$$c_{\text{LinearL1}}(y_{(a+1):b}) := \sum_{t=a+1}^{b} \|y_t - X_t\hat{\beta}\|_1$$

where $\hat{\beta}$ is fit column-by-column via IRLS on segment $(a+1):b$, iterated until convergence (`tol`) or `maxIter`. Unlike the other regression costs, this has no $O(1)$-per-segment closed form.

Returns: The segment cost.

Method `predict()`**:** Performs `binSeg` given a linear penalty value, or a target number of change-points.

Usage:

```
binSeg$predict(pen = 0, nBkps = NULL)
```

Arguments:

`pen` Numeric. Penalty per change-point. Ignored if `nBkps` is supplied. Default: `0`.

`nBkps` Integer. If supplied, takes precedence over `pen`: returns the first `nBkps` change-points `binSeg`'s greedy splitting added (see `$getHistory()`), i.e. its own best answer for that many change-points – not necessarily the globally optimal one for that count (see `Dynp` for that guarantee). Treated as an upper bound, not a strict requirement: if fewer than `nBkps` change-points were found (e.g. `minSize` leaves no further valid split), all of them are returned and a message reports the shortfall. Default: `NULL`.

Details: The algorithm recursively partitions a time series to detect multiple change-points. At each step, the algorithm identifies the segment that, if split, would result in the greatest reduction in total cost. This process continues until no further splits are possible (e.g., each segment is of minimal length or each breakpoint corresponds to a single data point).
Then, the algorithm selects the "optimal" set of break-points given the linear penalty threshold. Let $[c_1, \ldots, c_k, c_{k+1}]$ denote the set of segment end-points with $0 = c_0 < c_1 < c_2 < \cdots < c_k < c_{k+1} = n$, where k is the number of detected change-points and n is the total number of data points. and k is the number of change-points. Let $c_{(c_{i-1}, c_i]}$ be the cost of segment $(c_{i-1}, c_i]$. The total penalised cost is then

$$\text{TotalCost} = \sum_{i=1}^{k+1} c_{(c_{i-1}, c_i]} + \lambda \cdot k,$$

where λ is a linear penalty applied per change-point. We then optimise over k to minimise the penalised cost function.
This approach allows detecting multiple change-points in a time series while controlling model complexity through the linear penalty threshold.
In our implementation, the recursive step is carried out during `$fit()`. Therefore, `$predict()` runs in linear time with respect to the number of data points.
Temporary segment end-points are saved to `private$.tmpEndPoints` after `$predict()`, enabling users to call `$plot()` without specifying endpoints manually.

Returns: An integer vector of regime end-points. By design, the last element is the number of observations.

Method `segments()`**:** Returns the cost and parameter estimates of each segment from the latest `$predict()`.

Usage:

```
binSeg$segments()
```

Details: Let $0 = c_0 < c_1 < \cdots < c_{k+1} = n$ be the end-points from the latest `$predict()`, called with either `pen` or `nBkps`. Segment i is $(c_{i-1}, c_i]$, `Cost` is $c_{(c_{i-1}, c_i]}$ and `Params` is its minimiser. Both modes return the first k change-points of the greedy splitting path, so the costs sum to `$getHistory()$cost[k + 1]`. This is the cost of the greedy segmentation, not necessarily the minimal cost for k change-points (see `Dynp`).
Temporary end-points are cleared by `$fit()`, so `$predict()` must be run again after modifying the object via its active bindings.

Returns: A list with one element per segment $(Start, End]$. Each element is a list with:

`Start` Start index of the segment (exclusive, 0-based), same convention as `$eval()`.

`End` End index of the segment (inclusive).

`Cost` The segment cost, as returned by `$eval(Start, End)`.

`Params` A named list of the segment parameter estimates, e.g. `list(mean = ...)` for `"L2"`. See `costFactory` for the fields returned by each cost function.

Method `getHistory()`**:** Retrieves the full cost history and sequentially added breakpoints.

Usage:

```
binSeg$getHistory()
```

Returns: A `data.frame` with three columns:

`k` The number of change-points.

`cost` The total unpenalised cost of the segmentation.

`added_bkp` The breakpoint added at this step to achieve the cost.

Method `plotElbow()`**:** Plots the elbow curve (Total Cost vs. Number of Change-Points).

Usage:

```
binSeg$plotElbow(maxK = NULL)
```

Arguments:

`maxK` Integer. The maximum number of change-points to display on the plot. If `NULL`, displays the full history. Default: `NULL`.

Returns: A `ggplot` object.

Method `plot()`**:** Plots change-point segmentation

Usage:

```
binSeg$plot(
  d = 1L,
  endPts,
  dimNames,
  main,
  xlab,
  tsWidth = 0.25,
  tsCol = "#5B9BD5",
  bgCol = c("#A3C4F3", "#FBB1BD"),
  bgAlpha = 0.5,
  ncol = 1L
)
```

Arguments:

`d` Integer vector. Dimensions to plot. Default: `1L`.

`endPts` Integer vector. End points. Default: latest temporary changepoints obtained via `$predict()`.

`dimNames` Character vector. Feature names matching length of `d`. Defaults to `"X1", "X2", ...`.

`main` Character. Main title. Defaults to `"binSeg: d = ..."`.

`xlab` Character. X-axis label. Default: `"Time"`.

`tsWidth` Numeric. Line width for time series and segments. Default: `0.25`.

`tsCol` Character. Time series color. Default: `"#5B9BD5"`.

bgCol Character vector. Segment colors, recycled to length of endPts. Default: c("#A3C4F3", "#FBB1BD").

bgAlpha Numeric. Background transparency. Default: 0.5.

ncol Integer. Number of columns in facet layout. Default: 1L.

Details: Plots change-point segmentation results. Based on ggplot2. Multiple plots can easily be horizontally and vertically stacked using patchwork's operators / and |, respectively.

Returns: An object of classes gg and ggplot.

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
binSeg$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Author(s)

Minh Long Nguyen <edelweiss611428@gmail.com>
Toby Dylan Hocking <toby.hocking@r-project.org>
Charles Truong <ctruong@ens-paris-saclay.fr>
Huy Nhat Minh Nguyen <sleepysnorlax0115@gmail.com>

References

Truong, C., Oudre, L., & Vayatis, N. (2020). Selective review of offline change point detection methods. Signal Processing, 167, 107299.

Hocking, T. D. (2024). Finite Sample Complexity Analysis of Binary Segmentation. arXiv preprint arXiv:2410.08654.

Examples

```
## L2 example
set.seed(1121)
signals = as.matrix(c(rnorm(100,0,1),
                      rnorm(100,5,1)))
# Default L2 cost function
binSegObj = binSeg$new(minSize = 1L, jump = 1L)
binSegObj$fit(signals)
binSegObj$predict(pen = 100)
binSegObj$plot()

## SIGMA example
set.seed(111)
signals = as.matrix(c(rnorm(100,-5,1),
                      rnorm(100,-5,10),
                      rnorm(100,-5,1)))
# L2 cost function
binSegObj = binSeg$new(minSize = 1L, jump = 1L)
binSegObj$fit(signals)
```

```
# We choose pen = 50.
binSegObj$predict(pen = 50)
binSegObj$plot()

# The standard L2 cost function is not suitable.
# Use the SIGMA cost function.
binSegObj$costFunc = costFunc$new(costFunc = "SIGMA")
binSegObj$predict(pen = 50)
binSegObj$plot()
```

`costFactory` `costFactory` *class*

Description

An `R6` class for fast segment-cost evaluation and parameter estimation, without a detection algorithm.

Details

`costFactory` builds the `C++` cost module selected by `$costFunc` at `$fit()`, and keeps it, so every query reuses its precomputations, as `PELT`, `binSeg` and `Window` do. `$eval()` and `$get_params()` call the module directly: segments are `(a, b]` with 0-based `a`, and all checks are done in `C++`.
`$new()` only stores the `costFunc` object; `$fit()` validates and attaches the data. `$costFunc` is an active binding, so it can be inspected or replaced after construction – if data has already been supplied, replacing it automatically triggers `$fit()` again.
`$get_params()` returns:

- `"L1"`: `median`.
- `"L2"`: `mean`.
- `"SIGMA"`: `mean` and `cov` (plus `epsilon` on the diagonal if `addSmallDiag = TRUE`).
- `"VAR"`, `"LinearL2"` and `"LinearL1"`: `coef`, intercept first.
- `"LinearSIGMA"`: `coef` and `cov` (plus `epsilon` on the diagonal if `addSmallDiag = TRUE`).
- `"Custom"`: `params`, the output of `paramFun`; an empty list if `paramFun` is `NULL`.

Both `cov`s are biased maximum-likelihood estimates: divided by the segment length `b - a`, with no Bessel (`n - 1`) or degrees-of-freedom correction.

Methods

`$new()` Initialises a `costFactory` object.

`$fit()` Constructs the `C++` cost module.

`$eval()` Evaluates the cost of a segment.

`$get_params()` Returns the parameter estimates of a segment.

`$segments()` Returns the cost and parameter estimates of each segment of a given segmentation.

`$clone()` Clones the `R6` object.

Active bindings

`costFunc` R6 object of class `costFunc`. Can be accessed or modified via `$costFunc`. Modifying `costFunc` will automatically trigger `$fit()` if a `tsMat` has already been fitted.

Methods

Public methods:

- `costFactory$new()`
- `costFactory$fit()`
- `costFactory$eval()`
- `costFactory$get_params()`
- `costFactory$segments()`
- `costFactory$clone()`

Method `new()`**:** Initialises a `costFactory` object. Does not build the `C++` cost module; call `$fit()` for that.

Usage:

```
costFactory$new(costFunc)
```

Arguments:

`costFunc` A `R6` object of class `costFunc`. Should be created via `costFunc$new()` to avoid error. Default: `costFunc$new("L2")`.

Returns: Invisibly returns `NULL`.

Method `fit()`**:** Validates the supplied data and constructs the `C++` cost module selected by `$costFunc`.

Usage:

```
costFactory$fit(tsMat = NULL, covariates = NULL)
```

Arguments:

`tsMat` Numeric matrix. Time series of size $n \times p$. If `NULL`, the method will use the previously assigned `tsMat` (i.e., from a prior `$fit(tsMat)`). Default: `NULL`.

`covariates` Numeric matrix with `n` rows, used by `"LinearL2"`, `"LinearSIGMA"` and `"LinearL1"`. If `NULL` and no prior `covariates` were set, the model is force-fitted with only an intercept. Default: `NULL`.

Details: This method constructs the `C++` cost module and sets `private$.fitted` to `TRUE`, enabling the use of `$eval()` and `$get_params()`.

Returns: Invisibly returns `NULL`.

Method `eval()`**:** Evaluates the cost of the segment (a, b] with the module's `eval()`.

Usage:

```
costFactory$eval(a, b)
```

Arguments:

`a` Integer. Start index (exclusive, 0-based).

`b` Integer. End index (inclusive).

Returns: The segment cost.

Method `get_params()`**:** Returns the parameter estimates of the segment (a, b] with the module's `get_params()`.

Usage:

```
costFactory$get_params(a, b)
```

Arguments:

`a` Integer. Start index (exclusive, 0-based).

`b` Integer. End index (inclusive).

Returns: A named list; see Details.

Method `segments()`**:** Returns the cost and parameter estimates of each segment of a given segmentation.

Usage:

```
costFactory$segments(endPts)
```

Arguments:

`endPts` Integer vector. Segment end-points, e.g. from `$predict()` of `PELT`, `binSeg`, `Window` or `Dynp`. Sorted internally; must be unique, at least `1`, and end at `n`.

Returns: A list with one element per segment $(Start, End]$, in the same format as the `$segments()` of the segmentation classes. Each element is a list with `Start` (exclusive, 0-based), `End` (inclusive), `Cost` (as returned by `$eval(Start, End)`) and `Params` (as returned by `$get_params(Start, End)`).

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
costFactory$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Minh Long Nguyen <edelweiss611428@gmail.com>
Huy Nhat Minh Nguyen <sleepysnorlax0115@gmail.com>

Examples

```
set.seed(1)
tsMat = cbind(c(rnorm(100, 0), rnorm(100, 5, 5)))
cf = costFactory$new(costFunc$new("L2"))
cf$fit(tsMat)
cf$eval(0, 100)
cf$get_params(0, 100)
cf$segments(c(100, 200))

# `costFunc` is an active binding: swapping it re-fits automatically.
cf$costFunc = costFunc$new("SIGMA")
cf$eval(0, 100)
```

costFunc costFunc *class*

Description

An R6 class specifying a cost function

Details

Creates an instance of costFunc R6 class, used in initialisation of change-point detection modules. Currently supports the following cost functions:

- **L1 cost function**:

$$c_{L_1}(y_{(a+1)\dots b}) := \sum_{t=a+1}^{b} \|y_t - \tilde{y}_{(a+1)\dots b}\|_1$$

  where $\tilde{y}_{(a+1)\dots b}$ is the coordinate-wise median of the segment. If $a \geq b-1$, return 0.

- **L2 cost function**:

$$c_{L_2}(y_{(a+1)\dots b}) := \sum_{t=a+1}^{b} \|y_t - \bar{y}_{(a+1)\dots b}\|_2^2$$

  where $\bar{y}_{(a+1)\dots b}$ is the empirical mean of the segment. If $a \geq b-1$, return 0.

- **SIGMA cost function**:

$$c_{\Sigma}(y_{(a+1)\dots b}) := (b-a) \log\det \hat{\Sigma}_{(a+1)\dots b}$$

  where $\hat{\Sigma}_{(a+1)\dots b}$ is the empirical covariance matrix of the segment without Bessel's correction. Here, if addSmallDiag = TRUE, a small bias epsilon is added to the diagonal of estimated covariance matrices to improve numerical stability.

  By default, addSmallDiag = TRUE and epsilon = 1e-6. In case addSmallDiag = TRUE, if the covariance matrix is numerically singular (its log-determinant cannot be computed) or its log-determinant is smaller than the lower bound p*log(epsilon), return (b - a)*p*log(epsilon), otherwise, output an error message.

- **VAR(r) cost function**:

$$c_{\mathrm{VAR}}(y_{(a+1)\dots b}) := \sum_{t=\max(a,r)+1}^{b} \left\| y_t - \hat{c} - \sum_{j=1}^{r} \hat{A}_j y_{t-j} \right\|_2^2$$

  where $\hat{c}$ and $\hat{A}_j$ are the OLS estimates of the intercept and VAR coefficients on the segment. The lagged values y_{t-j} may come from before $a+1$, so only the first r observations of the whole series lack a full set of lags. If the system is singular, an approximate (minimum-norm) least-squares solve is used. If $b - a < p * r + 1$ (i.e., not enough observations), or $a > n - r$ (where n is the time series length), return 0.

- **"LinearL2"** for piecewise linear regression process with **constant noise variance**

$$c_{\text{LinearL2}}(y_{(a+1):b}) := \sum_{t=a+1}^{b} \|y_t - X_t\hat{\beta}\|_2^2$$

  where $\hat{\beta}$ are OLS estimates on segment $(a+1):b$. If segment is shorter than the minimum number of points needed for OLS, return 0.

- **"LinearSIGMA"** for piecewise linear regression process with **varying noise covariance**

$$c_{\text{LinearSIGMA}}(y_{(a+1):b}) := (b-a)\log\det\hat{\Sigma}_{(a+1):b}$$

  where $\hat{\Sigma}_{(a+1):b}$ is the empirical covariance matrix of OLS residuals $y - X\hat{\beta}$ on segment $(a+1):b$, divided by $b-a$ with no degrees-of-freedom correction for the fitted coefficients, and otherwise estimated the same way as in the SIGMA cost function (including the `addSmallDiag`/`epsilon` stabilisation and lower-bound fallback).

- **"LinearL1"** for piecewise linear regression process under **L1 (least absolute deviations) loss**

$$c_{\text{LinearL1}}(y_{(a+1):b}) := \sum_{t=a+1}^{b} \|y_t - X_t\hat{\beta}\|_1$$

  where $\hat{\beta}$ is fit column-by-column via Iteratively Reweighted Least Squares (IRLS), iterated until the change in the fit's cost is within `tol` or `maxIter` iterations are reached. Unlike the other regression cost functions, this has no $O(1)$-per-segment closed form, since IRLS must be re-run on each queried segment.

- **"Custom"** for a user-defined cost function supplied from R

$$c_{\text{Custom}}(y_{(a+1):b}) := \text{evalFun}(y_{(a+1):b}, a, b)$$

  where `evalFun` is a user-supplied function called on the raw segment matrix, plus the segment's own `(a,b]` bounds – the latter let `evalFun` align the segment against any externally-captured, position-indexed data (e.g. a weight vector or exogenous series `evalFun` closes over) without the package needing to know that data exists. Because each call crosses back into R, this is substantially slower per call than the built-in costs above; see `$evalFun` and `$paramFun`.

If active binding `$costFunc` is modified (via assignment operator), the default parameters will be used.

Methods

`$new()` Initialises a `costFunc` object.

`$pass()` Describes the `costFunc` object.

`$clone()` Clones the `costFunc` object.

Active bindings

`costFunc` Character. Cost function. Can be accessed or modified via `$costFunc`. If `costFunc` is modified and required parameters are missing, the default parameters are used.

`pVAR` Integer. Vector autoregressive order. Can be accessed or modified via `$pVAR`.

`addSmallDiag` Logical. Whether to add a bias value to the diagonal of estimated covariance matrices to stabilise matrix operations. Can be accessed or modified via `$addSmallDiag`.

`epsilon` Double. A bias value added to the diagonal of estimated covariance matrices to stabilise matrix operations. Can be accessed or modified via `$epsilon`.

`intercept` Logical. Whether to include the intercept in regression problems. Can be accessed or modified via `$intercept`.

`tol` Double. IRLS convergence tolerance: iteration stops once the change in the fit's cost falls below `tol`. Can be accessed or modified via `$tol`.

`maxIter` Integer. Maximum number of IRLS iterations. Can be accessed or modified via `$maxIter`.

`evalFun` Function. Required for `costFunc = "Custom"`. A user-defined cost function, called as `evalFun(segment, a, b)`, where `segment` is the numeric matrix of rows `(a+1):b` for the queried segment `(a,b]` (0-indexed, same convention as `$eval(a, b)`). `a` and `b` let `evalFun` align `segment` against externally-captured, position-indexed data it closes over (e.g. `externalSeries[(a+1):b]`), which the package itself never needs to see. Must return a single numeric value. Can be accessed or modified via `$evalFun`.

`paramFun` Function or `NULL`. Optional for `costFunc = "Custom"`. A user-defined function called as `paramFun(segment, a, b)` (same convention as `evalFun`), used by `$get_params()` to report segment-level estimates. If `NULL` (default), `$get_params()` returns an empty list for `"Custom"`. Can be accessed or modified via `$paramFun`.

Methods

Public methods:

- `costFunc$new()`
- `costFunc$pass()`
- `costFunc$clone()`

Method `new()`**:** Initialises a `costFunc` object.

Usage:

```
costFunc$new(costFunc, ...)
```

Arguments:

`costFunc` Character. Cost function. Supported values include `"L2"`, `"VAR"`, and `"SIGMA"`. Default: `L2`.

`...` Optional named parameters required by specific cost functions.
If any required parameters are missing or null, default values will be used.
For `"L1"` and `"L2"`, there is no extra parameter.
For `"SIGMA"`, supported parameters are:

`addSmallDiag` Logical. If `TRUE`, add a small value to the diagonal of estimated covariance matrices to stabilise matrix operations. Default: `TRUE`.

`epsilon` Double. If `addSmallDiag = TRUE`, a small positive value added to the diagonal of estimated covariance matrices to stabilise matrix operations. Default: `1e-6`.

For `"VAR"`, `pVAR` is required:

`pVAR` Integer. Vector autoregressive order. Must be a positive integer. Default: `1L`.

For "LinearL2", intercept is required:

intercept Logical. Whether to include the intercept in regression problems. Default: TRUE.

For "LinearSIGMA", supported parameters are:

intercept Logical. Whether to include the intercept in regression problems. Default: TRUE.

addSmallDiag Logical. If TRUE, add a small value to the diagonal of estimated residual covariance matrices to stabilise matrix operations. Default: TRUE.

epsilon Double. If addSmallDiag = TRUE, a small positive value added to the diagonal of estimated residual covariance matrices to stabilise matrix operations. Default: 1e-6.

For "LinearL1", supported parameters are:

intercept Logical. Whether to include the intercept in regression problems. Default: TRUE.

tol Double. IRLS convergence tolerance: iteration stops once the change in the fit's cost falls below tol. Default: 1e-6.

maxIter Integer. Maximum number of IRLS iterations. Default: 1000L.

For "Custom", supported parameters are:

evalFun Function. Required. See $evalFun for details.

paramFun Function or NULL. Optional. See $paramFun for details.

Method pass(): Returns a list of configuration parameters to initialise detection modules.

Usage:

```
costFunc$pass()
```

Method clone(): The objects of this class are cloneable with this method.

Usage:

```
costFunc$clone(deep = FALSE)
```

Arguments:

deep Whether to make a deep clone.

Author(s)

Minh Long Nguyen <edelweiss611428@gmail.com>

Examples

```
## L2 costFunc (default)
costFuncObj = costFunc$new()
costFuncObj$pass()
## SIGMA costFunc
costFuncObj = costFunc$new(costFunc = "SIGMA")
costFuncObj$pass()
# Modify active bindings
costFuncObj$epsilon = 10^-5
costFuncObj$pass()
costFuncObj$costFunc = "VAR"
costFuncObj$pass()
```

Dynp *Exact Dynamic Programming (*Dynp*)*

Description

An R6 class implementing exact dynamic programming for offline change-point detection.

Details

Dynp finds the segmentation that globally minimises the total cost for a *specified* number of change-points – unlike PELT (exact for a penalty, not a change-point count) and binSeg (greedy at each split, not guaranteed globally optimal for a fixed count), Dynp is exact for whatever nBkps is requested. The price is complexity: building the full table costs $O(\text{nBkpsMax} \cdot M^2)$ where M is the number of (minSize, jump)-admissible grid points ($M \approx n/\text{jump}$ in the worst case), versus PELT's near-linear pruning or binSeg's $O(n \log n)$-ish greedy search. $fit() computes the exact minimal cost for *every* change-point count from 0 to nBkpsMax in one pass (see $costPath()). Dynp is therefore useful when an exact fixed-number-of-change-points solution is required, particularly when using custom cost functions for which PELT pruning cannot be guaranteed.

Dynp requires a R6 object of class costFunc, exactly as PELT/binSeg/Window do – see costFunc for the supported cost functions ("L1", "L2", "SIGMA", "VAR", "LinearL2", "LinearSIGMA", "LinearL1", "Custom").

Some examples are provided below. See the package website for detailed usage!

Methods

$new() Initialises a Dynp object.

$describe() Describes the Dynp object.

$fit() Constructs a Dynp module in C++.

$eval() Evaluates the cost of a segment.

$predict() Returns the exact optimal breakpoints, for a given pen or nBkps.

$segments() Returns the cost and parameter estimates of each segment from the latest $predict().

$costPath() Returns the exact minimal cost for every change-point count up to nBkpsMax.

$getHistory() Same data as $costPath(), as a k/cost data.frame (same shape as binSeg/Window's $getHistory(), minus added_bkp – see its own docs for why).

$plotElbow() Plots the elbow curve (Total Cost vs. Number of Change-Points).

$plot() Plots change-point segmentation in ggplot style.

$clone() Clones the R6 object.

Active bindings

minSize Integer. Minimum allowed segment length. Can be accessed or modified via $minSize. Modifying minSize will automatically trigger $fit().

`jump` Integer. Search grid step size. Can be accessed or modified via `$jump`. Modifying `jump` will automatically trigger `$fit()`.

`nBkpsMax` Integer or `NULL`. Upper bound on the number of change-points the exact DP table is built for; `$predict(nBkps = ...)` treats this as a cap, not a requirement – see `$predict()`. If `NULL` (default), `$fit()` resolves it to `min(20, floor(n / minSize) - 1)`, i.e. the maximum feasible value given `minSize`, capped at 20. Set explicitly to go beyond 20 (up to the maximum feasible value) or to lower it; the table costs $O(\text{nBkpsMax} \cdot M^2)$ work and $O(\text{nBkpsMax} \cdot M)$ memory. Can be accessed or modified via `$nBkpsMax`; modifying it will automatically trigger `$fit()`.

`costFunc` `R6` object of class `costFunc`. Can be accessed or modified via `$costFunc`. Modifying `costFunc` will automatically trigger `$fit()`.

`tsMat` Numeric matrix. Input time series matrix of size $n \times p$. Can be accessed or modified via `$tsMat`. Modifying `tsMat` will automatically trigger `$fit()`.

`covariates` Numeric matrix. Input time series matrix having a similar number of observations as `tsMat`. Can be accessed or modified via `$covariates`. Modifying `covariates` will automatically trigger `$fit()`.

Methods

Public methods:

- `Dynp$new()`
- `Dynp$describe()`
- `Dynp$fit()`
- `Dynp$eval()`
- `Dynp$predict()`
- `Dynp$segments()`
- `Dynp$costPath()`
- `Dynp$getHistory()`
- `Dynp$plotElbow()`
- `Dynp$plot()`
- `Dynp$clone()`

Method `new()`**:** Initialises a `Dynp` object.

Usage:

```
Dynp$new(minSize, jump, nBkpsMax, costFunc)
```

Arguments:

`minSize` Integer. Minimum allowed segment length. Default: `1L`.

`jump` Integer. Search grid step size: only positions in {k, 2k, ...} are considered. Default: `1L`.

`nBkpsMax` Integer or `NULL`. Upper bound on the number of change-points to build the exact DP table for. Default: `NULL` (resolved to `min(20, floor(n / minSize) - 1)` at `$fit()` time).

`costFunc` A `R6` object of class `costFunc`. Should be created via `costFunc$new()` to avoid error. Default: `costFunc$new("L2")`.

Returns: Invisibly returns `NULL`.

Method `describe()`**:** Describes a `Dynp` object.

Usage:

```
Dynp$describe(printConfig = FALSE)
```

Arguments:

`printConfig` Logical. Whether to print object configurations. Default: `FALSE`.

Returns: Invisibly returns a list storing at least the following fields:

`minSize` Minimum allowed segment length.

`jump` Search grid step size.

`nBkpsMax` The user-set `nBkpsMax` (possibly `NULL`).

`resolvedNBkpsMax` The `nBkpsMax` actually used by the last `$fit()` (`NULL` if not fitted).

`costFunc` The `costFunc` object.

`fitted` Whether or not `$fit()` has been run.

`tsMat` Time series matrix.

`covariates` Covariate matrix (if exists).

`n` Number of observations.

`p` Number of features.

Method `fit()`**:** Constructs a `C++` module for `Dynp` and builds the exact DP table.

Usage:

```
Dynp$fit(tsMat = NULL, covariates = NULL)
```

Arguments:

`tsMat` Numeric matrix. A time series matrix of size $n \times p$ whose rows are observations ordered in time. If `tsMat = NULL`, the method will use the previously assigned `tsMat` (e.g., set via the active binding `$tsMat` or from a prior `$fit(tsMat)`). Default: `NULL`.

`covariates` Numeric matrix. A time series matrix having a similar number of observations as `tsMat`. Required for models involving both dependent and independent variables. If `covariates = NULL` and no prior covariates were set (i.e., `$covariates` is still `NULL`), the model is force-fitted with only an intercept. Default: `NULL`.

Details: This method constructs a `C++ Dynp` module and sets `private$.fitted` to `TRUE`, enabling the use of `$predict()`, `$costPath()` and `$eval()`. If `$nBkpsMax` is `NULL`, it is resolved here to `min(20, floor(n / minSize) - 1)` (with a message); if set higher than `floor(n / minSize) - 1`, it is capped to that (with a warning).

Returns: Invisibly returns `NULL`.

Method `eval()`**:** Evaluate the cost of the segment (a,b]

Usage:

```
Dynp$eval(a, b)
```

Arguments:

`a` Integer. Start index of the segment (exclusive). Must satisfy `start < end`.

`b` Integer. End index of the segment (inclusive).

Returns: The segment cost. See `costFunc` for the cost formulas.

Method `predict()`**:** Returns the exact optimal breakpoints, either under a linear penalty or for a specified number of change-points.

Usage:

```
Dynp$predict(pen = 0, nBkps = NULL)
```

Arguments:

`pen` Numeric. Penalty per change-point; the change-point count is chosen by minimising cost$+$ pen $\cdot\, k$ over $k = 0, \ldots,$ nBkpsMax (same convention as `PELT`/`binSeg`/`Window`). Ignored if `nBkps` is supplied. Default: `0`.

`nBkps` Integer. If supplied, takes precedence over `pen`: returns the exact optimal segmentation for exactly `nBkps` change-points. Treated as an upper bound, not a strict requirement, in the same spirit as `binSeg`/`Window`: if `nBkps` exceeds `nBkpsMax`, it is capped to `nBkpsMax` and a message reports the shortfall. Unlike `binSeg`/`Window`, the result is still exact for whatever count is actually used – capping only ever happens because the DP table wasn't built that far (a configuration choice via `$nBkpsMax`), not because `Dynp` ran out of candidates the way a greedy search can. If no valid segmentation exists at all for the (possibly capped) count – only possible when `jump` is large relative to `minSize` – this still errors, since there is no well-defined smaller-but-still-what-you-asked-for answer to fall back to. Default: `NULL`.

Details: With `nBkps = k`, this returns the segmentation into exactly `k` change-points that globally minimises the total cost – exact, unlike `binSeg$predict()` for the same `k`. With `pen`, this instead minimises the penalised cost over every change-point count the DP table covers, matching how `PELT`/`binSeg`/`Window` already select a count from a penalty.
Both the DP table (via `$fit()`) and the traceback here only depend on `minSize`/`jump` admissible positions, so `$predict()` itself is cheap (`O(nBkps)`) regardless of which mode is used – the cost was already paid in `$fit()`.
Temporary segment end-points are saved to `private$.tmpEndPoints` after `$predict()`, enabling users to call `$plot()` without specifying endpoints manually.

Returns: An integer vector of regime end-points. By design, the last element is the number of observations.

Method `segments()`**:** Returns the cost and parameter estimates of each segment from the latest `$predict()`.

Usage:

```
Dynp$segments()
```

Details: Let $0 = c_0 < c_1 < \cdots < c_{k+1} = n$ be the end-points from the latest `$predict()`. Segment i is $(c_{i-1}, c_i]$, `Cost` is $c_{(c_{i-1}, c_i]}$ and `Params` is its minimiser. The costs sum to `$costPath()[k + 1]`, the exact minimal total cost for k change-points. With `nBkps`, k is the requested count (capped at the resolved `nBkpsMax`). With `pen`, k is the count, from 0 up to the resolved `nBkpsMax`, that minimises the total cost plus `pen` times k.
Temporary end-points are cleared by `$fit()`, so `$predict()` must be run again after modifying the object via its active bindings.

Returns: A list with one element per segment $(Start, End]$. Each element is a list with:

`Start` Start index of the segment (exclusive, 0-based), same convention as `$eval()`.

`End` End index of the segment (inclusive).

`Cost` The segment cost, as returned by `$eval(Start, End)`.

`Params` A named list of the segment parameter estimates, e.g. `list(mean = ...)` for `"L2"`. See `costFactory` for the fields returned by each cost function.

Method `costPath()`**:** Returns the exact minimal cost for every change-point count the DP table covers.

Usage:

```
Dynp$costPath()
```

Details: This is the data behind the "elbow method" for choosing the number of change-points: plot it (see `$getHistory()`/`$plotElbow()`) and look for where the marginal decrease in cost flattens out. It comes for free out of `$fit()` – no extra computation is triggered here.

Returns: A numeric vector of length `resolvedNBkpsMax + 1`: element `k+1` is the exact minimal total cost of segmenting the series into exactly `k` change-points, for `k = 0, ..., resolvedNBkpsMax`. `Inf` at position `k+1` means no valid segmentation with exactly `k` change-points exists given `minSize`/`jump` (only possible when `jump` is large relative to `minSize`).

Method `getHistory()`**:** Retrieves the exact minimal cost for every change-point count the DP table covers, as a `data.frame` (same shape as `binSeg`/`Window`'s `$getHistory()`, minus `added_bkp`).

Usage:

```
Dynp$getHistory()
```

Details: Unlike `binSeg`/`Window`, there is no `added_bkp` column here: `binSeg` and `Window` each build a single *nested* sequence of change-points, where every `k`'s answer is the previous `k-1`'s answer plus one more point, so "the breakpoint added at this step" is well-defined. `Dynp`'s per-`k` solutions are each independently exact and need not be nested at all – the optimal segmentation for `k` change-points can differ completely from the one for `k-1`. Use `$predict(nBkps = k)` to get the full breakpoint set for a given `k`.

Returns: A `data.frame` with two columns:

`k` The number of change-points, from `0` to the resolved `nBkpsMax`.

`cost` The exact minimal total cost of segmenting the series into exactly `k` change-points (see `$costPath()`).

Method `plotElbow()`**:** Plots the elbow curve (Total Cost vs. Number of Change-Points).

Usage:

```
Dynp$plotElbow(maxK = NULL)
```

Arguments:

`maxK` Integer. The maximum number of change-points to display on the plot. If `NULL`, displays the full history. Default: `NULL`.

Returns: A `ggplot` object.

Method `plot()`**:** Plots change-point segmentation

Usage:

```
Dynp$plot(
  d = 1L,
  endPts,
  dimNames,
```

```
  main,
  xlab,
  tsWidth = 0.25,
  tsCol = "#5B9BD5",
  bgCol = c("#A3C4F3", "#FBB1BD"),
  bgAlpha = 0.5,
  ncol = 1L
)
```

Arguments:

`d` Integer vector. Dimensions to plot. Default: `1L`.

`endPts` Integer vector. End points. Default: latest temporary changepoints obtained via `$predict()`.

`dimNames` Character vector. Feature names matching length of `d`. Defaults to `"X1", "X2", ...`.

`main` Character. Main title. Defaults to `"Dynp: d = ..."`.

`xlab` Character. X-axis label. Default: `"Time"`.

`tsWidth` Numeric. Line width for time series and segments. Default: `0.25`.

`tsCol` Character. Time series color. Default: `"#5B9BD5"`.

`bgCol` Character vector. Segment colors, recycled to length of `endPts`. Default: `c("#A3C4F3", "#FBB1BD")`.

`bgAlpha` Numeric. Background transparency. Default: `0.5`.

`ncol` Integer. Number of columns in facet layout. Default: `1L`.

Details: Plots change-point segmentation results. Based on `ggplot2`. Multiple plots can easily be horizontally and vertically stacked using `patchwork`'s operators `/` and `|`, respectively.

Returns: An object of classes `gg` and `ggplot`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Dynp$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Minh Long Nguyen <edelweiss611428@gmail.com>
Toby Dylan Hocking <toby.hocking@r-project.org>
Charles Truong <ctruong@ens-paris-saclay.fr>
Huy Nhat Minh Nguyen <sleepysnorlax0115@gmail.com>

References

Truong, C., Oudre, L., & Vayatis, N. (2020). Selective review of offline change point detection methods. Signal Processing, 167, 107299.

Examples

```
## L2 example
set.seed(1121)
signals = as.matrix(c(rnorm(100,0,1),
                      rnorm(100,5,1)))
# Default L2 cost function; nBkpsMax left NULL -> resolved to min(20, maximum feasible value)
DynpObj = Dynp$new(minSize = 1L, jump = 1L)
DynpObj$fit(signals)
DynpObj$predict(nBkps = 1)
DynpObj$plot()

# Same result, reached via a penalty instead of an explicit count
DynpObj$predict(pen = 100)

# The exact cost of every change-point count from 0 to nBkpsMax -- elbow-method data
DynpObj$getHistory()
DynpObj$plotElbow()
```

PELT *Pruned Exact Linear Time (*PELT*)*

Description

An R6 class implementing the PELT algorithm for offline change-point detection.

Details

PELT (Pruned Exact Linear Time) is an efficient algorithm for change point detection that prunes the search space to achieve optimal segmentation in linear time under certain conditions.

PELT requires a R6 object of class costFunc, which can be created via costFunc$new(). Currently, the following cost functions are supported:

- "L1" and "L2" for (independent) piecewise Gaussian process with **constant variance**
- "SIGMA": for (independent) piecewise Gaussian process with **varying variance**
- "VAR": for piecewise Gaussian vector-regressive process with **constant noise variance**
- "LinearL2": for piecewise linear regression process with **constant noise variance**
- "LinearSIGMA": for piecewise linear regression process with **varying noise covariance**
- "LinearL1": for piecewise linear regression process under **L1 (least absolute deviations) loss**

See $eval() method for more details on computation of cost.

Some examples are provided below. See the package website for detailed usage!

Methods

`$new()` Initialises a `PELT` object.

`$describe()` Describes the `PELT` object.

`$fit()` Constructs a `PELT` module in `C++`.

`$eval()` Evaluates the cost of a segment.

`$predict()` Performs `PELT` given a linear penalty value.

`$segments()` Returns the cost and parameter estimates of each segment from the latest `$predict()`.

`$plot()` Plots change-point segmentation in `ggplot` style.

`$clone()` Clones the `R6` object.

Active bindings

`minSize` Integer. Minimum allowed segment length. Can be accessed or modified via `$minSize`. Modifying `minSize` will automatically trigger `$fit()`.

`jump` Integer. Search grid step size. Can be accessed or modified via `$jump`. Modifying `jump` will automatically trigger `$fit()`.

`costFunc` `R6` object of class `costFunc`. Search grid step size. Can be accessed or modified via `$costFunc`. Modifying `costFunc` will automatically trigger `$fit()`.

`tsMat` Numeric matrix. Input time series matrix of size $n \times p$. Can be accessed or modified via `$tsMat`. Modifying `tsMat` will automatically trigger `$fit()`.

`covariates` Numeric matrix. Input time series matrix having a similar number of observations as `tsMat`. Can be accessed or modified via `$covariates`. Modifying `covariates` will automatically trigger `$fit()`.

Methods

Public methods:

- `PELT$new()`
- `PELT$describe()`
- `PELT$fit()`
- `PELT$eval()`
- `PELT$predict()`
- `PELT$segments()`
- `PELT$plot()`
- `PELT$clone()`

Method `new()`**:** Initialises a `PELT` object.

Usage:

```
PELT$new(minSize, jump, costFunc)
```

Arguments:

`minSize` Integer. Minimum allowed segment length. Default: `1L`.

`jump` Integer. Search grid step size: only positions in {k, 2k, ...} are considered. Default: `1L`.

`costFunc` A `R6` object of class `costFunc`. Should be created via `costFunc$new()` to avoid error. Default: `costFunc$new("L2")`.

Returns: Invisibly returns `NULL`.

Method `describe()`**:** Describes a `PELT` object.

Usage:

```
PELT$describe(printConfig = FALSE)
```

Arguments:

`printConfig` Logical. Whether to print object configurations. Default: `FALSE`.

Returns: Invisibly returns a list storing at least the following fields:

`minSize` Minimum allowed segment length.

`jump` Search grid step size.

`costFunc` The `costFunc` object.

`fitted` Whether or not `$fit()` has been run.

`tsMat` Time series matrix.

`covariates` Covariate matrix (if exists).

`n` Number of observations.

`p` Number of features.

Method `fit()`**:** Constructs a `C++` module for `PELT`.

Usage:

```
PELT$fit(tsMat = NULL, covariates = NULL)
```

Arguments:

`tsMat` Numeric matrix. A time series matrix of size $n \times p$ whose rows are observations ordered in time. If `tsMat` = `NULL`, the method will use the previously assigned `tsMat` (e.g., set via the active binding `$tsMat` or from a prior `$fit(tsMat)`). Default: `NULL`.

`covariates` Numeric matrix. A time series matrix having a similar number of observations as `tsMat`. Required for models involving both dependent and independent variables. If `covariates` = `NULL` and no prior covariates were set (i.e., `$covariates` is still `NULL`), the model is force-fitted with only an intercept. Default: `NULL`.

Details: This method constructs a `C++ PELT` module and sets `private$.fitted` to `TRUE`, enabling the use of `$predict()` and `$eval()`.

Returns: Invisibly returns `NULL`.

Method `eval()`**:** Evaluate the cost of the segment (a,b]

Usage:

```
PELT$eval(a, b)
```

Arguments:

`a` Integer. Start index of the segment (exclusive). Must satisfy `start` < `end`.

`b` Integer. End index of the segment (inclusive).

Details: The segment cost is evaluated as follows:

- **L1 cost function**:

$$c_{L_1}(y_{(a+1)\dots b}) := \sum_{t=a+1}^{b} \|y_t - \tilde{y}_{(a+1)\dots b}\|_1$$

  where $\tilde{y}_{(a+1)\dots b}$ is the coordinate-wise median of the segment. If $a \geq b-1$, return 0.
- **L2 cost function**:

$$c_{L_2}(y_{(a+1)\dots b}) := \sum_{t=a+1}^{b} \|y_t - \bar{y}_{(a+1)\dots b}\|_2^2$$

  where $\bar{y}_{(a+1)\dots b}$ is the empirical mean of the segment. If $a \geq b-1$, return 0.
- **SIGMA cost function**:

$$c_{\Sigma}(y_{(a+1)\dots b}) := (b-a) \log\det \hat{\Sigma}_{(a+1)\dots b}$$

  where $\hat{\Sigma}_{(a+1)\dots b}$ is the empirical covariance matrix of the segment without Bessel's correction. Here, if `addSmallDiag = TRUE`, a small bias `epsilon` is added to the diagonal of estimated covariance matrices to improve numerical stability.

  By default, `addSmallDiag = TRUE` and `epsilon = 1e-6`. In case `addSmallDiag = TRUE`, if the covariance matrix is numerically singular (its log-determinant cannot be computed) or its log-determinant is smaller than the lower bound `p*log(epsilon)`, return `(b - a)*p*log(epsilon)`, otherwise, output an error message.
- **VAR(r) cost function**:

$$c_{\mathrm{VAR}}(y_{(a+1)\dots b}) := \sum_{t=\max(a,r)+1}^{b} \left\| y_t - \hat{c} - \sum_{j=1}^{r} \hat{A}_j y_{t-j} \right\|_2^2$$

  where $\hat{c}$ and $\hat{A}_j$ are the OLS estimates of the intercept and VAR coefficients on the segment. The lagged values y_{t-j} may come from before $a+1$, so only the first r observations of the whole series lack a full set of lags. If the system is singular, an approximate (minimum-norm) least-squares solve is used. If $b - a < p * r + 1$ (i.e., not enough observations), or $a > n - r$ (where `n` is the time series length), return 0.

"LinearL2" for piecewise linear regression process with **constant noise variance**

$$c_{\mathrm{LinearL2}}(y_{(a+1):b}) := \sum_{t=a+1}^{b} \|y_t - X_t\hat{\beta}\|_2^2$$

where $\hat{\beta}$ are OLS estimates on segment $(a+1) : b$. If segment is shorter than the minimum number of points needed for OLS, return 0.

- **"LinearSIGMA"** for piecewise linear regression process with **varying noise covariance**

$$c_{\mathrm{LinearSIGMA}}(y_{(a+1):b}) := (b-a) \log\det \hat{\Sigma}_{(a+1):b}$$

  where $\hat{\Sigma}_{(a+1):b}$ is the empirical covariance matrix of OLS residuals $y - X\hat{\beta}$ on segment $(a+1) : b$, divided by $b-a$ with no degrees-of-freedom correction for the fitted coefficients, and otherwise estimated the same way as in the SIGMA cost function.

- **"LinearL1"** for piecewise linear regression process under **L1 (least absolute deviations) loss**

$$c_{\text{LinearL1}}(y_{(a+1):b}) := \sum_{t=a+1}^{b} \|y_t - X_t\hat{\beta}\|_1$$

where $\hat{\beta}$ is fit column-by-column via IRLS on segment $(a+1) : b$, iterated until convergence (`tol`) or `maxIter`. Unlike the other regression costs, this has no $O(1)$-per-segment closed form.

Returns: The segment cost.

Method `predict()`**:** Performs `PELT` given a linear penalty value.

Usage:

```
PELT$predict(pen = 0)
```

Arguments:

`pen` Numeric. Penalty per change-point. Default: `0`.

Details: The `PELT` algorithm detects multiple change-points by finding the set of break-points that globally minimises a penalised cost function. `PELT` uses dynamic programming combined with a pruning rule to reduce the number of candidate change-points, achieving efficient computation.
Let $[c_1, \ldots, c_k, c_{k+1}]$ denote the set of segment end-points with $0 = c_0 < c_1 < c_2 < \cdots < c_k < c_{k+1} = n$, where k is the number of detected change-points and n is the total number of data points. Let $c_{(c_{i-1},c_i]}$ be the cost of segment $(c_{i-1}, c_i]$. The total penalised cost is

$$\text{TotalCost} = \sum_{i=1}^{k+1} c_{(c_{i-1},c_i]} + \lambda \cdot k,$$

where λ is a linear penalty applied per change-point. `PELT` finds the set of endpoints that minimises this cost exactly.
The pruning step eliminates candidate change-points that cannot lead to an optimal solution, allowing `PELT` to run in linear time with respect to the number of data points.
Temporary segment end-points are saved to `private$.tmpEndPoints` after `$predict()`, enabling users to call `$plot()` without specifying endpoints manually.

Returns: An integer vector of regime end-points. By design, the last element is the number of observations.

Method `segments()`**:** Returns the cost and parameter estimates of each segment from the latest `$predict()`.

Usage:

```
PELT$segments()
```

Details: Let $0 = c_0 < c_1 < \cdots < c_{k+1} = n$ be the end-points from the latest `$predict(pen)`. Segment i is $(c_{i-1}, c_i]$, `Cost` is $c_{(c_{i-1},c_i]}$ and `Params` is its minimiser. The costs therefore sum to the optimal total penalised cost minus $\lambda \cdot k$.
Temporary end-points are cleared by `$fit()`, so `$predict()` must be run again after modifying the object via its active bindings.

Returns: A list with one element per segment $(Start, End]$. Each element is a list with:

`Start` Start index of the segment (exclusive, 0-based), same convention as `$eval()`.

`End` End index of the segment (inclusive).

`Cost` The segment cost, as returned by `$eval(Start, End)`.

`Params` A named list of the segment parameter estimates, e.g. `list(mean = ...)` for `"L2"`. See `costFactory` for the fields returned by each cost function.

Method `plot()`**:** Plots change-point segmentation

Usage:

```
PELT$plot(
  d = 1L,
  endPts,
  dimNames,
  main,
  xlab,
  tsWidth = 0.25,
  tsCol = "#5B9BD5",
  bgCol = c("#A3C4F3", "#FBB1BD"),
  bgAlpha = 0.5,
  ncol = 1L
)
```

Arguments:

`d` Integer vector. Dimensions to plot. Default: `1L`.

`endPts` Integer vector. End points. Default: latest temporary changepoints obtained via `$predict()`.

`dimNames` Character vector. Feature names matching length of `d`. Defaults to `"X1", "X2", ...`.

`main` Character. Main title. Defaults to `"PELT: d = ..."`.

`xlab` Character. X-axis label. Default: `"Time"`.

`tsWidth` Numeric. Line width for time series and segments. Default: `0.25`.

`tsCol` Character. Time series color. Default: `"#5B9BD5"`.

`bgCol` Character vector. Segment colors, recycled to length of `endPts`. Default: `c("#A3C4F3", "#FBB1BD")`.

`bgAlpha` Numeric. Background transparency. Default: `0.5`.

`ncol` Integer. Number of columns in facet layout. Default: `1L`.

Details: Plots change-point segmentation results. Based on `ggplot2`. Multiple plots can easily be horizontally and vertically stacked using `patchwork`'s operators `/` and `|`, respectively.

Returns: An object of classes `gg` and `ggplot`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
PELT$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Minh Long Nguyen <edelweiss611428@gmail.com>
Toby Dylan Hocking <toby.hocking@r-project.org>
Charles Truong <ctruong@ens-paris-saclay.fr>
Huy Nhat Minh Nguyen <sleepysnorlax0115@gmail.com>

References

Truong, C., Oudre, L., & Vayatis, N. (2020). Selective review of offline change point detection methods. Signal Processing, 167, 107299.

Killick, R., Fearnhead, P., & Eckley, I. A. (2012). Optimal detection of change points with a linear computational cost. Journal of the American Statistical Association, 107(500), 1590-1598.

Examples

```
## L2 example
set.seed(1121)
signals = as.matrix(c(rnorm(100,0,1),
                      rnorm(100,5,1)))
# Default L2 cost function
PELTObj = PELT$new(minSize = 1L, jump = 1L)
PELTObj$fit(signals)
PELTObj$predict(pen = 100)
PELTObj$plot()

## SIGMA example
set.seed(111)
signals = as.matrix(c(rnorm(100,-5,1),
                      rnorm(100,-5,10),
                      rnorm(100,-5,1)))
# L2 cost function
PELTObj = PELT$new(minSize = 1L, jump = 1L)
PELTObj$fit(signals)
# We choose pen = 50.
PELTObj$predict(pen = 50)
PELTObj$plot()

# The standard L2 cost function is not suitable.
# Use the SIGMA cost function.
PELTObj$costFunc = costFunc$new(costFunc = "SIGMA")
PELTObj$predict(pen = 50)
PELTObj$plot()
```

Window *Slicing Window (*Window*)*

Description

An R6 class implementing slicing window for offline change-point detection.

Details

Slicing window is a scalable, linear-time change-point detection algorithm that selects breakpoints based on local gains computed over sliding windows.

Currently supports the following cost functions:

- "L1" and "L2" for (independent) piecewise Gaussian process with **constant variance**
- "SIGMA": for (independent) piecewise Gaussian process with **varying variance**
- "VAR": for piecewise Gaussian vector-regressive process with **constant noise variance**
- "LinearL2": for piecewise linear regression process with **constant noise variance**
- "LinearSIGMA": for piecewise linear regression process with **varying noise covariance**
- "LinearL1": for piecewise linear regression process under **L1 (least absolute deviations) loss**

Window requires a R6 object of class costFunc, which can be created via costFunc$new(). Currently, the following cost functions are supported:

- "L1" and "L2" for (independent) piecewise Gaussian process with **constant variance**
- "SIGMA": for (independent) piecewise Gaussian process with **varying variance**
- "VAR": for piecewise Gaussian vector-regressive process with **constant noise variance**
- "LinearL2": for piecewise linear regression process with **constant noise variance**
- "LinearSIGMA": for piecewise linear regression process with **varying noise covariance**
- "LinearL1": for piecewise linear regression process under **L1 (least absolute deviations) loss**

See $eval() method for more details on computation of cost.

Some examples are provided below. See the package website for detailed usage!

Methods

$new() Initialises a Window object.

$describe() Describes the Window object.

$fit() Constructs a Window module in C++.

$eval() Evaluates the cost of a segment.

$predict() Performs Window given a linear penalty value.

$segments() Returns the cost and parameter estimates of each segment from the latest $predict().

$getHistory() Retrieves the full cost history and sequentially added breakpoints.

$plotElbow() Plots the elbow curve (Total Cost vs. Number of Change-Points).

$plot() Plots change-point segmentation in ggplot style.

$clone() Clones the R6 object.

Active bindings

`minSize` Integer. Minimum allowed segment length. Can be accessed or modified via `$minSize`. Modifying `minSize` will automatically trigger `$fit()`.

`radius` Integer. Window radius. Can be accessed or modified via `$radius`. Modifying `radius` will automatically trigger `$fit()`.

`jump` Integer. Search grid step size. Can be accessed or modified via `$jump`. Modifying `jump` will automatically trigger `$fit()`.

`costFunc` `R6` object of class `costFunc`. Search grid step size. Can be accessed or modified via `$costFunc`. Modifying `costFunc` will automatically trigger `$fit()`.

`tsMat` Numeric matrix. Input time series matrix of size $n \times p$. Can be accessed or modified via `$tsMat`. Modifying `tsMat` will automatically trigger `$fit()`.

`covariates` Numeric matrix. Input time series matrix having a similar number of observations as `tsMat`. Can be accessed or modified via `$covariates`. Modifying `covariates` will automatically trigger `$fit()`.

Methods

Public methods:

- `Window$new()`
- `Window$describe()`
- `Window$fit()`
- `Window$eval()`
- `Window$predict()`
- `Window$segments()`
- `Window$getHistory()`
- `Window$plotElbow()`
- `Window$plot()`
- `Window$clone()`

Method `new()`**:** Initialises a `Window` object.

Usage:

```
Window$new(minSize, jump, radius, costFunc)
```

Arguments:

`minSize` Integer. Minimum allowed segment length. Default: `1L`.

`jump` Integer. Search grid step size: only positions in {k, 2k, ...} are considered. Default: `1L`.

`radius` Integer. Radius of each sliding window. Default: `1L`.

`costFunc` A `R6` object of class `costFunc`. Should be created via `costFunc$new()` to avoid error. Default: `costFunc$new("L2")`.

Returns: Invisibly returns `NULL`.

Method `describe()`**:** Describes a `Window` object.

Usage:

```
Window$describe(printConfig = FALSE)
```

Arguments:

`printConfig` Logical. Whether to print object configurations. Default: `FALSE`.

Returns: Invisibly returns a list storing at least the following fields:

`minSize` Minimum allowed segment length.

`jump` Search grid step size.

`radius` Radius of each sliding window.

`costFunc` The `costFunc` object.

`fitted` Whether or not `$fit()` has been run.

`tsMat` Time series matrix.

`covariates` Covariate matrix (if exists).

`n` Number of observations.

`p` Number of features.

Method `fit()`**:** Constructs a `C++` module for `Window`.

Usage:

```
Window$fit(tsMat = NULL, covariates = NULL)
```

Arguments:

`tsMat` Numeric matrix. A time series matrix of size $n \times p$ whose rows are observations ordered in time. If `tsMat = NULL`, the method will use the previously assigned `tsMat` (e.g., set via the active binding `$tsMat` or from a prior `$fit(tsMat)`). Default: `NULL`.

`covariates` Numeric matrix. A time series matrix having a similar number of observations as `tsMat`. Required for models involving both dependent and independent variables. If `covariates = NULL` and no prior covariates were set (i.e., `$covariates` is still `NULL`), the model is force-fitted with only an intercept. Default: `NULL`..

Details: This method constructs a `C++ Window` module and sets `private$.fitted` to `TRUE`, enabling the use of `$predict()` and `$eval()`. Some precomputations are performed to allow `$predict()` to run in linear time with respect to the number of local change-points (see `$predict()` for more details).

Returns: Invisibly returns `NULL`.

Method `eval()`**:** Evaluate the cost of the segment (a,b]

Usage:

```
Window$eval(a, b)
```

Arguments:

`a` Integer. Start index of the segment (exclusive). Must satisfy `start < end`.

`b` Integer. End index of the segment (inclusive).

Details: The segment cost is evaluated as follows:

- **L1 cost function**:

$$c_{L_1}(y_{(a+1)\dots b}) := \sum_{t=a+1}^{b} \|y_t - \tilde{y}_{(a+1)\dots b}\|_1$$

where $\tilde{y}_{(a+1)\dots b}$ is the coordinate-wise median of the segment. If $a \geq b - 1$, return 0.

- **L2 cost function**:

$$c_{L_2}(y_{(a+1)\dots b}) := \sum_{t=a+1}^{b} \|y_t - \bar{y}_{(a+1)\dots b}\|_2^2$$

  where $\bar{y}_{(a+1)\dots b}$ is the empirical mean of the segment. If $a \geq b-1$, return 0.
- **SIGMA cost function**:

$$c_{\Sigma}(y_{(a+1)\dots b}) := (b-a)\log\det\hat{\Sigma}_{(a+1)\dots b}$$

  where $\hat{\Sigma}_{(a+1)\dots b}$ is the empirical covariance matrix of the segment without Bessel's correction. Here, if `addSmallDiag = TRUE`, a small bias `epsilon` is added to the diagonal of estimated covariance matrices to improve numerical stability.

  By default, `addSmallDiag = TRUE` and `epsilon = 1e-6`. In case `addSmallDiag = TRUE`, if the covariance matrix is numerically singular (its log-determinant cannot be computed) or its log-determinant is smaller than the lower bound `p*log(epsilon)`, return `(b - a)*p*log(epsilon)`, otherwise, output an error message.
- **VAR(r) cost function**:

$$c_{\mathrm{VAR}}(y_{(a+1)\dots b}) := \sum_{t=\max(a,r)+1}^{b} \left\| y_t - \hat{c} - \sum_{j=1}^{r} \hat{A}_j y_{t-j} \right\|_2^2$$

  where $\hat{c}$ and $\hat{A}_j$ are the OLS estimates of the intercept and VAR coefficients on the segment. The lagged values y_{t-j} may come from before $a+1$, so only the first r observations of the whole series lack a full set of lags. If the system is singular, an approximate (minimum-norm) least-squares solve is used. If $b - a < p * r + 1$ (i.e., not enough observations), or $a > n - r$ (where `n` is the time series length), return 0.
- **"LinearL2"** for piecewise linear regression process with **constant noise variance**

$$c_{\mathrm{LinearL2}}(y_{(a+1):b}) := \sum_{t=a+1}^{b} \|y_t - X_t\hat{\beta}\|_2^2$$

  where $\hat{\beta}$ are OLS estimates on segment $(a+1):b$. If segment is shorter than the minimum number of points needed for OLS, return 0.
- **"LinearSIGMA"** for piecewise linear regression process with **varying noise covariance**

$$c_{\mathrm{LinearSIGMA}}(y_{(a+1):b}) := (b-a)\log\det\hat{\Sigma}_{(a+1):b}$$

  where $\hat{\Sigma}_{(a+1):b}$ is the empirical covariance matrix of OLS residuals $y - X\hat{\beta}$ on segment $(a+1):b$, divided by $b-a$ with no degrees-of-freedom correction for the fitted coefficients, and otherwise estimated the same way as in the SIGMA cost function.
- **"LinearL1"** for piecewise linear regression process under **L1 (least absolute deviations) loss**

$$c_{\mathrm{LinearL1}}(y_{(a+1):b}) := \sum_{t=a+1}^{b} \|y_t - X_t\hat{\beta}\|_1$$

  where $\hat{\beta}$ is fit column-by-column via IRLS on segment $(a+1):b$, iterated until convergence (`tol`) or `maxIter`. Unlike the other regression costs, this has no $O(1)$-per-segment closed form.

Returns: The segment cost.

Method `predict()`**:** Performs `Window` given a linear penalty value, or a target number of change-points.

Usage:

```
Window$predict(pen = 0, nBkps = NULL)
```

Arguments:

`pen` Numeric. Penalty per change-point. Ignored if `nBkps` is supplied. Default: `0`.

`nBkps` Integer. If supplied, takes precedence over `pen`: returns the `nBkps` highest-gain local maxima found by `$fit()` (see `$getHistory()`), i.e. `Window`'s own best answer for that many change-points – not necessarily the globally optimal one for that count (see `Dynp` for that guarantee). Treated as an upper bound, not a strict requirement: `Window` only ever finds finitely many local maxima, so if fewer than `nBkps` exist, all of them are returned and a message reports the shortfall. Default: `NULL`.

Details: The algorithm scans the data with a fixed-size window to detect candidate local change-points (lcps) if the gains of its k_{thresh} neighbors to the left and right are all smaller than its gain, where k_{thresh} is defined as

$$k_{\text{thresh}} = \max\left(\left\lfloor \frac{\max(2\text{radius}, 2 \cdot \text{minSize})}{2 \cdot \text{jump}} \right\rfloor, 1\right)$$

After candidate local change-points and computing the local gains, the algorithm selects the "optimal" set of break-points given the linear penalty threshold. Let G_i denote the local gain for candidate change-point i, for $i = 1, \ldots, n_{\text{lcps}}$. The local gains are ordered such that $G_1 \geq G_2 \geq \cdots \geq G_{n_{\text{lcps}}}$. Note that it is possible that no local change-points are detected, for example if the window size is too large.
The total cost for the selected k change-points is then calculated as

$$\text{TotalCost} = -\sum_{i=1}^{k} G_i + \lambda \cdot k,$$

where λ is a linear penalty applied per change-point. We then optimise over $k = 1, \ldots, n_{\text{lcps}}$ to minimise the penalised cost function. $k = 0$ is not a candidate, so at least one change-point is returned whenever a local change-point exists, however large the penalty.
This approach allows detecting multiple change-points in a time series while controlling model complexity through the linear penalty threshold.
In our implementation, scanning the data to detect candidate local change-points and computing their corresponding local gains is already performed in `$fit()`. Therefore, `$predict()` runs in linear time with respect to the number of local change-points.
Temporary segment end-points are saved to `private$.tmpEndPoints` after `$predict()`, enabling users to call `$plot()` without specifying endpoints manually.

Returns: An integer vector of regime end-points. By design, the last element is the number of observations.

Method `segments()`**:** Returns the cost and parameter estimates of each segment from the latest `$predict()`.

Usage:

```
Window$segments()
```

Details: Let $0 = c_0 < c_1 < \cdots < c_{k+1} = n$ be the end-points from the latest `$predict()`, called with either `pen` or `nBkps`. Segment i is $(c_{i-1}, c_i]$, `Cost` is $c_{(c_{i-1},c_i]}$ and `Params` is its minimiser. The costs sum to the total cost of this segmentation. For $k \geq 1$ this generally differs from `$getHistory()$cost[k + 1]`, which subtracts local gains computed on windows of `2 * radius` points instead of re-evaluating the segments.
Temporary end-points are cleared by `$fit()`, so `$predict()` must be run again after modifying the object via its active bindings.

Returns: A list with one element per segment $(Start, End]$. Each element is a list with:

`Start` Start index of the segment (exclusive, 0-based), same convention as `$eval()`.

`End` End index of the segment (inclusive).

`Cost` The segment cost, as returned by `$eval(Start, End)`.

`Params` A named list of the segment parameter estimates, e.g. `list(mean = ...)` for `"L2"`. See `costFactory` for the fields returned by each cost function.

Method `getHistory()`**:** Retrieves the full cost history and sequentially added breakpoints.

Usage:

```
Window$getHistory()
```

Returns: A `data.frame` with three columns:

`k` The number of change-points.

`cost` The total unpenalised cost of the segmentation.

`added_bkp` The breakpoint added at this step to achieve the cost.

Method `plotElbow()`**:** Plots the elbow curve (Total Cost vs. Number of Change-Points).

Usage:

```
Window$plotElbow(maxK = NULL)
```

Arguments:

`maxK` Integer. The maximum number of change-points to display on the plot. If `NULL`, displays the full history. Default: `NULL`.

Returns: A `ggplot` object.

Method `plot()`**:** Plots change-point segmentation

Usage:

```
Window$plot(
  d = 1L,
  endPts,
  dimNames,
  main,
  xlab,
  tsWidth = 0.25,
  tsCol = "#5B9BD5",
  bgCol = c("#A3C4F3", "#FBB1BD"),
  bgAlpha = 0.5,
  ncol = 1L
)
```

Arguments:

`d` Integer vector. Dimensions to plot. Default: `1L`.

`endPts` Integer vector. End points. Default: latest temporary changepoints obtained via `$predict()`.

`dimNames` Character vector. Feature names matching length of `d`. Defaults to `"X1", "X2", ...`.

`main` Character. Main title. Defaults to `"Window: d = ..."`.

`xlab` Character. X-axis label. Default: `"Time"`.

`tsWidth` Numeric. Line width for time series and segments. Default: `0.25`.

`tsCol` Character. Time series color. Default: `"#5B9BD5"`.

`bgCol` Character vector. Segment colors, recycled to length of `endPts`. Default: `c("#A3C4F3", "#FBB1BD")`.

`bgAlpha` Numeric. Background transparency. Default: `0.5`.

`ncol` Integer. Number of columns in facet layout. Default: `1L`.

Details: Plots change-point segmentation results. Based on `ggplot2`. Multiple plots can easily be horizontally and vertically stacked using `patchwork`'s operators `/` and `|`, respectively.

Returns: An object of classes `gg` and `ggplot`.

Method `clone()`**:** The objects of this class are cloneable with this method.

Usage:

```
Window$clone(deep = FALSE)
```

Arguments:

`deep` Whether to make a deep clone.

Author(s)

Minh Long Nguyen <edelweiss611428@gmail.com>
Toby Dylan Hocking <toby.hocking@r-project.org>
Charles Truong <ctruong@ens-paris-saclay.fr>
Huy Nhat Minh Nguyen <sleepysnorlax0115@gmail.com>

References

Truong, C., Oudre, L., & Vayatis, N. (2020). Selective review of offline change point detection methods. Signal Processing, 167, 107299.

Examples

```
## L2 example
set.seed(1121)
signals = as.matrix(c(rnorm(100,0,1),
                      rnorm(100,5,1)))
# Default L2 cost function
WindowObj = Window$new(minSize = 1L, jump = 1L)
WindowObj$fit(signals)
WindowObj$predict(pen = 100)
WindowObj$plot()

## SIGMA example
```

```
set.seed(111)
signals = as.matrix(c(rnorm(100,-5,1),
                      rnorm(100,-5,10),
                      rnorm(100,-5,1)))
# L2 cost function
WindowObj = Window$new(minSize = 1L, jump = 1L)
WindowObj$fit(signals)
# We choose pen = 50.
WindowObj$predict(pen = 50)
WindowObj$plot()

# The standard L2 cost function is not suitable.
# Use the SIGMA cost function.
WindowObj$costFunc = costFunc$new(costFunc = "SIGMA")
WindowObj$predict(pen = 50)
WindowObj$plot()
```

Index

binSeg, 2

costFactory, 9
costFunc, 12

Dynp, 16

PELT, 22

Window, 28